Y Combinator,
CC/CK machine,
and continuations

CS245 — Spring 2019

Authors: Kris Micinski + Thomas Gilray
\[
e ::= (\text{letrec} ([x (\lambda (x) e)]) ) \\
  \text{|} (\lambda (x) e) \\
  \text{|} (e e) \\
  \text{|} x \\
\]

\[
x ::= \text{<vars>} 
\]
Infinity
Ω

(((\(x\) \((x\ x)\)) \((x\ x)\))\)
Key: U takes a function and calls it on itself

\[ (((\lambda \ (x) \ (x \ x)) \ (\lambda \ (x) \ (x \ x))) \ (\lambda \ (x) \ (x \ x))) \]
(define U (\ (f) (f f)))

(letrec ([fib (lambda (x) (if (= x 0) 1 (* x (fib (- x 1)))))])
 (fib 3))

(let ([fib (U (lambda (f)
                (lambda (x) (if (= x 0) 1 (* x (... (- x 1)))))))]
 (fib 3))

What can I type right here to make fib work?

(Hint: the answer can be written in 5 characters)
(define U (\f\) (f f)))

(letrec ([fib (lambda (x) (if (= x 0) 1 (* x (fib (- x 1))))))]
(fib 3))

(let ([fib (U (lambda (f)
(lambda (x) (if (= x 0) 1 (* x (... (- x 1)))))))]
(fib 3))

What can I type right here to make fib work?

(f f)
Y combinator
(letrec ([fact (λ (n) 
  (if (= n 0) 
    1 
    (* n (fact (- n 1))))))))

(fact 5)
Key idea: instead of

(let ([mk (λ (mk) (λ (n)
  (if (= n 0)
      1
      (* n ((mk mk) (- n 1))))))]
  ((mk mk) 5)))
$(Y \ f) = f \ (Y \ f)$

(It’s a fixed-point combinator!)
Three step process for deriving $Y$

$$(Y \ f) = f \ (Y \ f)$$

$Y = (\lambda \ (f) \ (f \ (Y \ f)))$ \hspace{1cm} 1. Treat as definition

$mY = (\lambda \ (mY) \ \\
\hspace{1cm} (\lambda \ (f) \ \\
\hspace{2cm} (f \ ((mY \ mY) \ f))))$ \hspace{1cm} 2. Lift to mk-$Y$,

use self-application

$mY = (\lambda \ (mY) \ \\
\hspace{1cm} (\lambda \ (f) \ \\
\hspace{2cm} (f \ (\lambda \ (x) (((mY \ mY) \ f) \ x)))))$ \hspace{1cm} 3. Eta-expand
\[ Y = (U \ (\lambda (y) \ (\lambda (f) \ (f \ (\lambda (x) (((y \ y) \ f) \ x)))))) \]

**U-combinator:** \((U \ U)\) is Omega
(let ([fact (Y (λ (fact) (λ (n) (if (= n 0) 1 (* n (fact (- n 1)))))))])

(fact 5))
Try an example!!!

(define Y (((λ (x) (x x)) (λ (y) (λ (f)
                (f (λ (x) (((y y) f) x)))))))

(define (fib x)
  (if (or (= x 0) (= x 1))
      1
      (+ (fib (- x 1)) (fib (- x 2))))))

Rewrite this to use the Y combinator instead
e ::= \ (\lambda\ (x)\ e) \\
| \ (e\ e) \\
| x
De-churching

(define (church->nat cv)
  )

(define (church->list cv)
  )

(define (church->bool cv)
  )
De-churching

(define (church->nat cv)
  ((cv add1) 0))

(define (church->list cv)
  )

(define (church->bool cv)
  )
De-churching

(define (church->nat cv)
  ((cv add1) 0))

(define (church->list cv)
  ((cv (λ (car)
        (λ (cdr)
          (cons car
                (church->list cdr))))
       (λ (na) ‘())))

(define (church->bool cv)
  )
De-churching

(define (church->nat cv)
  ((cv add1) 0))

(define (church->list cv)
  ((cv (λ (car)
        (λ (cdr)
          (cons car
              (church->list cdr))))
    (λ (na) `())))

(define (church->bool cv)
  ((cv (λ () #t))
    (λ () #f)))
(letrec ([map (λ (f lst)
  (if (null? lst)
    '()
    (cons (f (car lst))
      (map f (cdr lst)))))]

  (map (λ (x) (+ 1 x))
    '(0 5 3))))
(define lst
 (((((((((
   (λ (Y-comb)
   (λ (church:null?)
   (λ (church:cons)
   (λ (church:car)
   (λ (church:cdr)
   (λ (church:+)
   (λ (church:*)
   (λ (church:not)
   ((λ (map)
     ((map
       (λ (x)
       ((church:+ (λ (f) (λ (x) (f x)))
        x)))
      ((church:cons (λ (f) (λ (x) x)))
       ((church:cons
         (λ (f)
         (λ (x) (f (f (f (f (f x)))))))))
      ((church:cons
        (λ (f) (λ (x) (f (f (f (f (f x))))))))))
      (λ (when-cons)
       (λ (when-null)
        (when-null (λ (x) x)))))))))))))
   (Y-comb
    (λ (map))
   (map
    (λ (x)
      ((church:+ (λ (f) (λ (x) (f x)))
       x)))
    ((church:cons (λ (f) (λ (x) x)))
     ((church:cons
       (λ (f)
       (λ (x) (f (f (f (f (f x)))))))))
     ((church:cons
       (λ (f) (λ (x) (f (f (f (f (f x))))))))))
     (λ (when-cons)
      (λ (when-null)
       (when-null (λ (x) x)))))))))))
)

> (map church->nat (church->list lst))
'(1 6 4)
Abstract Machine Zoo

C Term-rewriting Machine
Evaluation contexts

Restrict the order in which we may simplify a program’s redexes

\[ \mathcal{E} ::= (\mathcal{E} \ e) \]
\[ \quad | \quad (v \ \mathcal{E}) \]
\[ \quad | \quad \square \]

(left-to-right) CBV evaluation

\[ v ::= (\lambda (x) \ e) \]
\[ e ::= (\lambda (x) \ e) \]
\[ \quad | \quad (e \ e) \]
\[ \quad | \quad x \]

(left-to-right) CBN evaluation
Context and redex

\[
\mathcal{E} = \Box (\lambda (w) w)
\]

For CBV a redex must be \((v v)\)
For CVN, a redex must be \((v e)\)

\[
\mathcal{E}[ (v v) ] = ((\lambda (x) ((\lambda (y) y) x)) (\lambda (z) z)) (\lambda (w) w))
\]

\[
r = ((\lambda (x) ((\lambda (y) y) x)) (\lambda (z) z))
\]
Context and redex

$$\mathcal{E}[r] =$$

$$((\lambda (x) ((\lambda (y) y) x)) (\lambda (z) z)) (\lambda (w) w))$$

$$\mathcal{E} = (\Box (\lambda (w) w))$$

$$r = ((\lambda (x) ((\lambda (y) y) x)) (\lambda (z) z))$$

$$\rightarrow_\beta ((\lambda (y) y) (\lambda (z) z))$$
Put the reduced redex back in its evaluation context...

\[ \mathcal{E} = (\Box (\lambda (w) w)) \]

\[ r = (((\lambda (x)) ((\lambda (y) y) x)) (\lambda (z) z)) \]

\[ \rightarrow_{\beta} (((\lambda (y) y) (\lambda (z) z)) \]

\[ \mathcal{E}[r] \]

\[ (((\lambda (y) y) (\lambda (z) z)) (\lambda (w) w)) \]
Exercises—can you evaluate...

1) \(((\lambda (y) y) (\lambda (z) z)) (\lambda (w) w))\)

2) \(((\lambda (u) (u u)) (\lambda (x) (\lambda (x) x)))\)

3) \(((\lambda (x) x) (\lambda (y) y)) (\lambda (u) (u u)) (\lambda (z) (z z)))\)
Abstract Machine Zoo

C  Term-rewriting Machine
CC  Context and Redex Machine
CK  Control / Continuation Machine

Next time…
Continuations: first-class control
Continuations

A *continuation* is a return point, a call stack, or the remainder of the program, viewed as a function.

In Scheme, continuations are first-class values that can be captured using the language form `call/cc` and passed around to be invoked later.
First-class continuations

We may consider several alternative viewpoints on first-class continuations:

A **continuation** is a value encoding a **saved return point** to resume.

A **continuation** is a function encoding the **remainder of the program**.

A **continuation** is a function that never returns. When invoked on an input value, it resumes a previous return point with that value, and finishes the program from that return point until it exits.

**Continuations** generalize all known control constructs: gotos, loops, return statements, exceptions, C’s `longjmp`, threads/coroutines, etc.
Continuations are said to permit **time travel**, in the sense that they permit jumping back to a saved dynamic **evaluation context**: a previous call stack, or point in time, or a **future** one!

This may shorten the stack

...grow the stack

...or replace it entirely.
\[(\text{call/cc } e_0)\]

\text{call with current continuation}

call/cc takes a single argument, a callback, which it applies on the \textit{current continuation}—that is, the return from call/cc as a first-class function that saves the full call stack under call/cc.
Takes the call stack at the second argument expression of (+ ...) and saves it, essentially as a function, bound to k, that can be invoked on a value for that expression at a later point in time.

When k is invoked on the number 2, execution jumps back to the saved return point for call/cc and returns 2, returning 3 from the program as a whole.

(+ 1 (call/cc (lambda (k) (k 2))))
;; => 3
The program never returns from call \((k \ 2)\) because *undelimited continuations* run until the program exits.

call/cc gives us undelimited (a.k.a. full) continuations.

\[
(+ \ 1 \ (\text{call/cc} \ (\lambda (k) \ (k \ 2))))
\]

;; => 3

\[
(+ \ 1 \ (\text{call/cc} \ (\lambda (k) \ (k \ 2) \ (\text{print } 0))))
\]

;; => 3  (print 0) is never reached
(+ 1 (call/cc (lambda (k) (k 2))))
;; => 3

This call/cc’s behavior is *roughly* the same as the application:

```scheme
((lambda (k) (k 2))
 (lambda (n) (exit (print (+ 1 n))))))
;; => 3
```

Where the high-lit continuation `(lambda (n) ...)` takes a return value for the `(call/cc ...)` expression and finishes the program.
A common idiom for call/cc is to let-bind the current continuation.

(let ([cc (call/cc (lambda (k) k))])

...)

A common idiom for call/cc is to let-bind the current continuation.
Note that applying call/cc on the identity function is exactly the same as applying it on the u-combinator!

Why is this the case?
call/cc makes a tail call to \((\text{lambda} \ (k) \ ...))\), so the body of the function is the same return point as the captured continuation \(k\)!

\[
\begin{align*}
\text{(let ([(cc (call/cc (lambda (k) ...)))])} \\
\quad \ldots) \quad \uparrow \quad \uparrow
\end{align*}
\]

This return point \(\ldots\) is the same as this one\(\ldots\)

\[
\begin{align*}
\text{(let ([(cc (call/cc (lambda (k) (k k)))])])} \\
\quad \ldots) \quad \uparrow
\end{align*}
\]

\(\ldots\) and calling \(k\) on itself, returns \(k\) to itself!

Returning value \(v\) is the same as \textit{calling} that saved return point \textit{on} \(v\).
Continuations can be used to jump back to a previous point.

Just as we could have invoked call/cc on the u-combinator, to jump back to the let-binding of cc, returning cc, we call (cc cc).
(define (fun x)

  (let ( [[y (if (p? x)
            ...
            ...)]] )

    (g x y)))

A simple use of continuations is to implement a 
**preemptive return**.

What if we wanted to return from fun within the right-hand-side of the let form?
Binds the return-point of the current call to `fun` to a continuation `return`.

```scheme
(define (fun x)
  (call/cc (lambda (return)
    (let ([y (if (p? x)
              ...
              (return x))])
      (g x y))))
```

Uses the continuation `return` to jump back to the return point of `fun` and yield value `x` instead of binding `y` and calling `g`. 
Try an example. What do each of these 3 examples return?  
(Hint: Racket evaluates argument expressions left to right.)

\[
\begin{align*}
\text{(call/cc (lambda (k0))} & \quad (+ 1 \text{(call/cc (lambda (k1))}) \\
& \quad (+ 1 (k0 3))))))
\end{align*}
\]

\[
\begin{align*}
\text{(call/cc (lambda (k0))} & \quad (+ 1 \text{(call/cc (lambda (k1))}) \\
& \quad (+ 1 (k0 (k1 3))))))
\end{align*}
\]

\[
\begin{align*}
\text{(call/cc (lambda (k0))} & \quad (+ 1 \\
& \quad \text{(call/cc (lambda (k1))}) \\
& \quad (+ 1 (k1 3)))) \\
& \quad (k0 1)))
\end{align*}
\]
Try an example. What do each of these 3 examples return? (Hint: Racket evaluates argument expressions left to right.)

(call/cc (lambda (k0)
    (+ 1 (call/cc (lambda (k1)
        (+ 1 (k0 3))))))))

(call/cc (lambda (k0)
    (+ 1 (call/cc (lambda (k1)
        (+ 1 (k0 (k1 3))))))))

(call/cc (lambda (k0)
    (+ 1 (call/cc (lambda (k1)
        (+ 1 (k1 3))))))
    (k0 1))))

3

4

1
Continue and break

A Python `while` loop on the left that supports `continue` and `break` can be implemented using `call/cc` as the Scheme on the right.

```scheme
(call/cc (λ (break)
  (letrec ([loop (λ ()
    (when cond
      (call/cc (λ (continue)
        body))
    (loop))])
  (loop)))))
```

```python
while cond:
  body
else:
  otherwise
```
Continuations and mutation

(let* ([n 2]
        [cc (call/cc (lambda (k) k))])
  (set! n (+ n 1))
  (if (<= n 4)
    (cc cc)
    n))

Does this program terminate? What does it return?
Continuations and mutation

(let* ([n 2]
        [cc (call/cc (lambda (k) k))]
        (set! n (+ n 1))
        (if (<= n 4)
            (cc cc)
            n))

This loop terminates and returns 5.

This illustrates that invoking a continuation resumes a previous call stack, but does not revert mutations—changes made in the heap.
Try an example. What do each of these 2 examples return?
(Hint: Racket evaluates argument expressions left to right.)

```
(define n 3)
(+ n (call/cc
    (lambda (cc)
      (set! n (+ n 1))
      (cc 1)))))
```

```
(define n 3)
(+ (call/cc
    (lambda (cc)
      (set! n (+ n 1))
      (cc 1)))
    n)
```
Try an example. What do each of these 2 examples return? (Hint: Racket evaluates argument expressions left to right.)

```
(define n 3)
(+ n (call/cc
    (lambda (cc)
      (set! n (+ n 1))
      (cc 1))))
```

4

```
(define n 3)
(+ (call/cc
    (lambda (cc)
      (set! n (+ n 1))
      (cc 1)))
n)
```

5
Stack-passing (CEK) semantics
(implementing first-class continuations)
C  Control-expression
   Term-rewriting / textual reduction
   Context and redex for deterministic eval

CE  Control & Env machine
    Big-step, explicit closure creation

CES Store-passing machine
    Passes addr->value map in evaluation order

CEK Stack-passing machine
    Passes a list of stack frames, small-step
\((e_0, \text{env}) \Downarrow ((\lambda (x) e_2), \text{env'})\) \hspace{1em} (e_1, \text{env}) \Downarrow v_1 \hspace{1em} (e_2, \text{env'}[x \mapsto v_1]) \Downarrow v_2

\((e_0 \ e_1, \text{env}) \Downarrow v_2\)

\((\lambda (x) e), \text{env}) \Downarrow ((\lambda (x) e), \text{env})\)

\((x, \text{env}) \Downarrow \text{env}(x)\)
Previously…

\[(e_0, e_1), \text{ env} \quad \rightarrow \quad v\]
Previously…

$$(e_0, e_1), \text{ env } v$$
(define (interp e env)
  (match e
    [([? symbol? x)
      (hash-ref env x)]

    [(`(\(,x) ,e_0)
      `(clo (\(,x) ,e_0) ,env)]

    [`(\(,e_0 ,e_1)
      (define v_0 (interp e_0 env))
      (define v_1 (interp e_1 env))
      (match v_0
        [(`(clo (\(,x) ,e_2) ,env)
          (interp e_2 (hash-set env x v_1)))]))])
\[ e ::= (\lambda (x) \, e) \]
\[ | \ (e \ e) \]
\[ | \ x \]
\[ | \ (\text{call/cc} \ (\lambda (x) \, e)) \]
\[ k ::= \text{halt} \mid \text{ar}(e, \text{env}, k) \mid \text{fn}(v, k) \]

\[ e ::= (\lambda (x) e) \mid (e \ e) \mid x \mid (\text{call/cc} (\lambda (x) e)) \]
\[ k ::= \text{halt} \mid \text{ar}(e, \text{env}, k) \mid \text{fn}(v, k) \]

\[ e ::= (\lambda (x) \ e) \]
\[ \mid (e \ e) \]
\[ \mid x \]
\[ \mid (\text{call/cc} \ (\lambda (x) \ e)) \]

\[ \mathcal{E} ::= (\mathcal{E} \ e) \]
\[ \mid (v \ \mathcal{E}) \]
\[ \mid \square \]
\(((e_0 \ e_1), \text{env}, k) \rightarrow (e_0, \text{env}, \text{ar}(e_1, \text{env}, k))\)

\((x, \text{env}, \text{ar}(e_1, \text{env}_1, k_1)) \rightarrow (e_1, \text{env}_1, \text{fn}(\text{env}(x), k_1))\)

\(((\lambda \ (x) \ e), \text{env}, \text{ar}(e_1, \text{env}_1, k_1)) \rightarrow (e_1, \text{env}_1, \text{fn}(((\lambda \ (x) \ e), \text{env}), k_1))\)

\((x, \text{env}, \text{fn}(((\lambda \ (x_1) \ e_1), \text{env}_1), k_1)) \rightarrow (e_1, \text{env}_1[x_1 \mapsto \text{env}(x)], k_1)\)

\(((\lambda \ (x) \ e), \text{env}, \text{fn}(((\lambda \ (x_1) \ e_1), \text{env}_1), k_1)) \\
\quad \quad \rightarrow (e_1, \text{env}_1[x_1 \mapsto ((\lambda \ (x) \ e), \text{env})], k_1)\)
call/cc semantics

$$((\text{call/cc } (\lambda (x) \, e_\theta)), \, \text{env}, \, k) \rightarrow (e_\theta, \, \text{env}[x \mapsto k], \, k)$$

$$((\lambda (x) \, e_\theta), \, \text{env}, \, \text{fn}(k_0, \, k_1)) \rightarrow ((\lambda (x) \, e_\theta), \, \text{env}, \, k_0)$$

$$(x, \, \text{env}, \, \text{fn}(k_0, \, k_1)) \rightarrow (x, \, \text{env}, \, k_0)$$
e ::= ... | (let ([x e₀]) e₁)

k ::= ... | let(x, e, env, k)

(x, env, let(x₁, e₁, env₁, k₁)) → (e₁, env₁[x₁ ↦ env(x)], k₁)

((\(x\) e), env, let(x₁, e₁, env₁, k₁)) → (e₁, env₁[x₁ ↦ (\(x\) e), env]], k₁)
These are nearly identical because a let form is just an immediate application of a lambda!

\[
\begin{align*}
(x, \text{env}, \text{fn}(((\lambda (x_1) e_1), \text{env}_1), k_1)) \rightarrow (e_1, \text{env}_1[x_1 \mapsto \text{env}(x)], k_1) \\
((\lambda (x) e), \text{env}, \text{fn}(((\lambda (x_1) e_1), \text{env}_1), k_1)) \\
\rightarrow (e_1, \text{env}_1[x_1 \mapsto ((\lambda (x) e), \text{env})], k_1)
\end{align*}
\]
CEK-machine evaluation

\[(e_\theta, [], ()) \rightarrow \ldots \]
\[\ldots \]
\[\ldots \]
\[\ldots \]
\[\ldots \]
\[\rightarrow (x, \text{env}, \text{halt}) \rightarrow \text{env}(x)\]
consider the following question.

Is it possible to take an arbitrary Racket/Scheme program and transform it systematically so that no function ever returns?